

Corona: A High Performance Publish-Subscribe System for the World Wide Web

Venugopalan Ramasubramanian Ryan Peterson Emin Gün Sirer
Department of Computer Science, Cornell University, Ithaca, NY 14853
{ramasv,ryanp,egs}@cs.cornell.edu

Abstract

Despite the abundance of frequently changing information, the Web lacks a publish-subscribe interface for delivering updates to clients. The use of naive polling for update detection leads to poor performance and limits scalability, as clients do not detect updates quickly and servers face high loads imposed by active polling. This paper describes Corona, a publish-subscribe system for the Web that provides high performance and scalability through optimal resource allocation. Users register interest in web pages through existing instant messaging services. Corona monitors the subscribed web pages, detects updates efficiently by allocating polling load among cooperating peers and disseminates them quickly to the clients. A distributed optimization engine ensures that Corona achieves the best update performance without exceeding load limits on content servers. Large scale simulations and measurements from Planet-Lab deployment, described in this paper, demonstrate that Corona achieves orders of magnitude improvement in update performance at a modest cost.

1 Introduction

Even though web content changes rapidly, existing web protocols do not provide a mechanism for automatically notifying users of updates. The growing popularity of frequently updated content, such as weblogs, collaboratively authored web pages (wikis), and news sites, motivates a *publish-subscribe* mechanism that can deliver updates to users quickly and efficiently, with low aggregate load on the network and content providers. Further, the recent emergence of micronews syndication mechanisms based on naive, repeated polling indicates that backwards compatibility with existing web tools and protocols is also critical for rapid adoption.

Publish-subscribe through uncoordinated polling, similar to the current micronews syndication, suffers from poor performance and scalability. Subscribers do not receive updates quickly, as the polling period poses a fundamental limit to the update detection time. Clients are tempted to poll at faster rates in order to detect updates quickly. Consequently, content providers have to handle the high bandwidth load imposed by clients, each

polling independently and multiple times for the same content. Moreover, the workload tends to be “sticky;” that is, users subscribed to popular content do not unsubscribe after their interest diminishes, causing a large amount of wasted bandwidth. Existing micronews syndication systems provide ad hoc, stop-gap measures to these problems. Content providers currently impose hard rate-limits based on IP addresses, which render the system inoperable for users sharing an IP address, or they provide hints for when not to poll, which are discretionary and imprecise. The fundamental problem is that the server bandwidth is used inefficiently, and stems from an architecture based on naive, uncoordinated polling.

This paper describes a novel distributed system for detecting and disseminating updates to web pages. Our system, called Corona, provides a high-performance update notification service for the Web without requiring any changes to the existing infrastructure, such as web servers. Corona enables any client to subscribe for updates to any existing web page or micronews feed, and asynchronously and efficiently delivers updates. The key contribution that enables such a general and backwards-compatible system is a distributed, peer-to-peer, cooperative resource management framework that can determine the optimal amount of bandwidth to devote to polling data sources in order to meet system-wide goals.

The key resource tradeoff in a publish-subscribe system where publishers are exogenous entities that serve content only when polled involves bandwidth versus update latency. Clearly, polling data sources more frequently will enable the system to detect and disseminate updates earlier. Yet polling every data source constantly would place a large burden on publishers, congest the network, and potentially run afoul of server-imposed polling limits that would ban the system from monitoring the micronews feed or web page. The goal of Corona, then, is to maximize the effective benefit of the aggregate bandwidth available to the system, while remaining within server-imposed bandwidth limits. Corona resolves the fundamental tradeoff between bandwidth and update latency by expressing it formally as an optimization problem and solving it to achieve either minimal update latency subject to a bandwidth limit or minimal bandwidth consumption for a targeted average update la-

tency.

The optimal allocation of bandwidth in Corona is computed using a decentralized algorithm that works on top of a distributed peer-to-peer overlay. Corona takes channel popularity, update rate, content size, and internal system overhead stemming from accounting and dissemination of meta-information into account when making bandwidth allocation decisions. While similar in spirit to the Beehive system for optimal replication [23], the Corona approach is fundamentally different in that the problem is significantly more complex as it takes many more parameters into account, the solution strategy is not purely analytical but numerical, and the fundamental resource tradeoff is entirely different.

The distributed bandwidth allocation algorithm that powers Corona can be executed to optimize the system for different performance goals and resource limits. In this paper, we examine two relevant setups: how to minimize update latency while ensuring that the average load on publishers is no more than what it would have been without Corona, and how to minimize bandwidth consumption in order to achieve a targeted update latency. We also examine variants of these two main approaches where the load is more fairly balanced across channels.

The front-end client interface to Corona is through existing instant messaging (IM) services. Users subscribe for content by sending instant messages to a registered Corona IM handle, and receive update notifications asynchronously. Internally, Corona consists of a cloud of nodes that monitor the set of active feeds or web pages called *channels*. The Corona resource allocation algorithm determines the number of nodes designated to monitor each channel. Cooperative polling ensures that the system can detect updates quickly while no single node exceeds server-designated limits on polling frequency. Each server dedicated to monitoring a channel has a copy of the latest version of the channel contents. A feed-specific *difference engine* determines whether detected changes are germane by filtering out superficial differences such as timestamps and advertisements, extracts the relevant portions that have changed, and distributes the delta-encoded changes to all internal nodes assigned to monitor the channel, which in turn distribute it to subscribed clients via IM.

We have implemented a prototype of Corona and deployed it on Planet-Lab. Evaluation of this deployment shows that Corona achieves one to three orders of magnitude improvement in update performance. In experiments parameterized by real RSS workload collected at Cornell [19] and spanning 80 Planet-Lab nodes and involving 30,000 subscriptions for 3000 different channels, Corona clients see fresh updates in intervals of 64 sec on average compared to legacy RSS clients, which see a mean update interval of 15 min. At all times during the

experiment, Corona issues no more polling requests to the content servers than issued by the legacy RSS clients.

Overall, Corona is a new overlay-based publish-subscribe system for the Web that provides asynchronous notifications, fast update detection, and optimal bandwidth utilization. This paper makes three contributions: (i) it outlines the general design of a publish-subscribe system that does not require any changes to content sources, (ii) formalizes the tradeoffs as an optimization problem and presents a novel, distributed numerical solution technique for determining the allocation of bandwidth that will achieve globally targeted goals while respecting resource limits, and (iii) presents results from extensive simulations and a live deployment that demonstrate that the system is practical.

The rest of the paper is organized as follows. The next section provides background on publish-subscribe systems and discusses other related work. Section 1 describes the architecture of Corona in detail. Implementation details are presented in Section 4 and experimental results based on simulations and deployment are described in Section 5. Finally, Section 6 summarizes our contributions and concludes.

2 Background and Related Work

Publish-subscribe systems have raised considerable interest in the research community over the years. In this section, we provide background on publish-subscribe based content distribution and summarize the current state of the art.

Publish-Subscribe Systems: The publish-subscribe paradigm consists of three components: *publishers*, who generate and feed the content into the system, *subscribers*, who specify content of their interest, and an infrastructure for matching subscriber interests with published content and delivering matched content to the subscribers. Based on the expressiveness of subscriber interests, Pub-sub systems can be classified as *topic-based* or *content-based*. In topic-based systems, publishers and subscribers are connected together by pre-defined topics, called *channels*; content is published on well-advertised channels to which users subscribe to and receive asynchronous updates. Content-based systems enable subscribers to express elaborate queries on the content and use sophisticated content filtering techniques to match subscriber interests with published content.

Prior research on pub-sub systems has primarily focused on the design and implementation of content filtering and event delivery mechanisms. Topic-based publish-subscribe systems have been built based on several decentralized mechanisms, such as group communication in Isis [13], shared object spaces in Linda [5] and TSpace [31], and rendezvous points in TIBCO [30]

and Herald [4]. Content-based publish-subscribe systems that use in network content filtering and aggregation include SIENA [6], Gryphon [29], and Astrolabe [32]. YFilter [8], Quark [3], and XTreeNet [11] are recent architectures proposed for supporting complex content-based queries on semi-structured XML data.

The fundamental drawback of the preceding publish-subscribe systems is their non-compatibility with the current Web architecture. They require substantial changes in the way publishers serve content, expect subscribers to learn sophisticated query languages, or propose to layout middle-boxes in the core of the Internet. On the other hand, Corona not only interoperates with the current pull-based Web architecture, but also complements it. It requires no changes to legacy web servers and provides an easy-to-use IM based interface to the users. Optimal resource management in Corona aimed at bounding network load insulates web servers from high load during flash-crowds.

Micronews Systems: Micronews feeds are short descriptions of frequently updated information, such as news stories and blog updates, in XML based formats such as RSS [26] and Atom [1]. They are accessed via HTTP through URLs and supported by client applications and browser plug-ins called *feed readers*, which check the contents of micronews feeds periodically and automatically on the user's behalf and display the returned results. The micronews standards envision a publish-subscribe mode of content dissemination and define XML tags such as *cloud* that tell clients how to receive asynchronous updates as well as *ttl*, *SkipHours*, and *SkipDays* that inform clients when not to poll. Yet, few content providers currently use the *cloud* tag to deliver asynchronous updates.

Recently, commercial services have started disseminating micronews updates through instant messages [14]. While Corona also uses IM for disseminating updates, it differs fundamentally from these commercial services, which use centralized servers and relentless polling to detect updates. Corona is layered on a self-organizing overlay comprised of cooperative peers that share updates. It is completely distributed and does not depend on centralized infrastructure or administration.

FeedTree [27], is a recently proposed system for disseminating micronews feeds that also uses a structured overlay and shares updates between peers. FeedTree nodes decide to poll for a feed and share updates based in an ad hoc manner based on heuristics. Corona's key contribution is the use of informed tradeoffs to optimal resource management. This principled approach enables Corona to provide the best update performance for its users, while ensuring that content servers are lightly loaded and do not get overwhelmed due to flash-crowds or sticky-traffic.

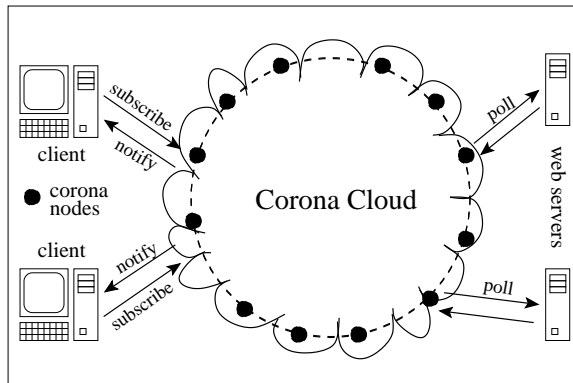


Figure 1: **Corona Architecture:** Corona is a distributed publish-subscribe system for the Web. It detects Web updates by polling cooperatively and notifies clients through instant messaging.

Overlay Networks: Corona is layered on structured overlays and leverages the underlying structure to facilitate optimal resource management. Recent years has seen a large number of structured overlays that organize the network based on finger-tables [28], hyper-dimensional cubes [24], rings with prefix-based routing [25, 34, 21], butterfly structure [20], de-Bruijn graphs [17, 33], or skip-lists [15]. Corona is agnostic about the choice of the overlay and can be easily layered on any overlay with uniform node degree, including the ones listed here.

Corona's approach to a peer-to-peer resource management problem has a similar flavor to that of Beehive [23], a structured replication framework that resolves space-time tradeoffs optimizations in structured overlays. Corona differs fundamentally from Beehive in two ways. First, the Beehive problem domain is limited to object replication in systems where objects have homogeneous popularity, size and update rate properties, whereas Corona is designed for the web environment where such properties can vary by several orders of magnitude between objects [10, 19], and thus takes them into account during optimization. Second, the more complex optimization problem renders the Beehive solution technique, based on mathematical derivation, fundamentally unsuitable for the problem tackled by Corona. Hence, Corona employs a more general and sophisticated numerical algorithm to perform its optimizations.

3 Corona

Corona (Cornell Online News Aggregator) is a topic-based publish-subscribe system for the Web. It provides asynchronous update notifications to clients, while interoperating with the current pull-based architecture of the Web. URLs of Web content serve as topics or

channels in Corona; users register their interest in some Web content by providing its URL and receive updates asynchronously about changes posted to that URL. Any web object identifiable by a URL can be monitored with Corona. In the background, Corona checks for updates on registered channels by cooperatively polling the content servers from geographically distributed nodes.

We envisage Corona as an infrastructure service offered by a set of widely-distributed nodes. These nodes may be all part of the same administrative domain, such as Akamai, or consist of server-class nodes contributed by participating institutions. By participating in Corona, institutions can significantly reduce the network bandwidth consumed in frequent redundant polling for content updates, as well as reduce the peak loads seen at content providers that they themselves may host. Corona nodes self-organize to form a structured overlay system. We use structured overlays to organize the distributed system as they are well known to provide good failure-resilience, high scalability, and bounded worst-case delays [28, 25, 34, 24, 9, 15, 17, 21, 22, 33]. Figure 1 illustrates the overall architecture of Corona.

The central feature that enables Corona to achieve fast update detection is *cooperative polling*. Corona assigns multiple nodes to periodically poll for the same channel and shares updates detected by any polling node. In general, n nodes polling with the same polling interval and randomly distributed polling times can detect updates n times faster if they share updates with each other. While it is tempting to take the maximum advantage of cooperative polling by having every Corona node poll for every feed, such a naive approach is clearly unscalable and ends up imposing huge network load on both Corona and content servers.

Corona makes informed-decisions on distributing polling tasks among nodes. The number of nodes that poll for each channel is determined based on an analysis of the fundamental tradeoff between update performance and network load. Corona poses this tradeoff as an optimization problem and obtains the optimal solution using Honeycomb, a light-weight toolkit for computing optimal performance-overhead tradeoffs in structured distributed systems. This principled approach enables Corona to efficiently resolve the tradeoff between performance and scalability.

In this section, we provide detailed descriptions of the components of Corona’s architecture, including the analytical models, the optimization framework, update detection and notification mechanisms, and the user interface.

3.1 Analytical Modeling

Corona’s approach of analysis driven cooperative polling can be easily applied on any distributed system organized

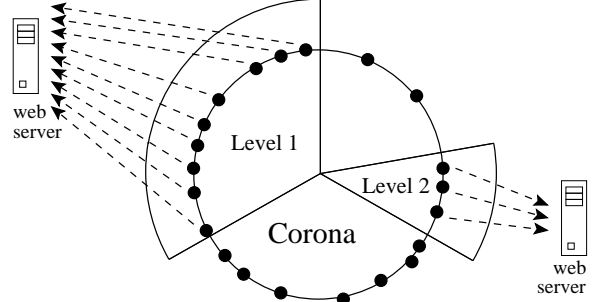


Figure 2: **Cooperative Polling in Corona:** Each channel is assigned a wedge of nodes to poll the content servers and detect updates. Corona determines the optimal wedge size for each channel through analysis of the global performance-overhead tradeoff.

as a structured overlay with uniform node degree. In this paper, we describe Corona using Pastry as the underlying substrate.

Pastry organizes the network into a ring by assigning identifiers from a circular numeric space to each node. The identifiers are treated as a sequence of digits of base b . In addition to neighbors along the ring, each node maintains contact with nodes that have matching prefix digits. These long-distance contacts are represented in a tabular structure called *routing table*. The entry in the i^{th} row and j^{th} column of the routing table points to a node whose identifier has the same i prefix digits as this node’s identifier and j as the $(i + 1)^{th}$ prefix digit. Essentially, the routing table defines a directed acyclic graph (DAG) rooted at each node, which can reach any other node in $\log_b N$ hops.

Corona assigns nodes in well-defined wedges of the Pastry ring for polling each channel. A wedge is defined by the number of matching prefix digits shared by the nodes with channel identifiers assigned from the same circular numeric space. A channel with *polling level* l is polled by all nodes with l matching prefix digits in their identifiers. The polling level 0 indicates that all the nodes in the system poll for the channel. Figure 2 illustrates the concept of polling levels in Corona.

Assigning well-defined portions of the ring enables Corona to manage polling efficiently with little overhead. The set of nodes polling for a channel can be represented by just a single number, the polling level, eliminating the expensive $O(n)$ complexity for managing state about cooperating nodes. Moreover, this also facilitates efficient update sharing, as a wedge is a subset of the DAG rooted at each node and all the nodes in a wedge can be reached quickly using the contacts in the routing table.

The polling level of a channel quantifies its performance-overhead tradeoff. A channel at level l has, on average, $\frac{N}{b^l}$ nodes polling it, which can cooperatively

detect updates in about $\frac{\tau}{2} \frac{b^i}{N}$ time on average, where τ is the polling interval. We estimate the average update detection time at a single node polling periodically at an interval τ to be $\frac{\tau}{2}$. Simultaneously, the collective load placed on the content server of this channel is $\tau \frac{N}{b^i}$. Note that we do not include the propagation delay for sharing updates in this analysis; this is because updates can be detected by comparing against any old version of the content. Hence, even if an update detected at a different node in the system is received late, the time to detect the next update at the current node does not change.

An easy way to set polling levels is to independently pick a level for each channel based on these estimates. However, such an approach involves investigating heuristics for determining the appropriate performance requirement for each channel and for dividing the total load between different channels. It does not provide a fine-grained control of the overall performance of the system and can end up operating at a point far from the global optimal. The rest of this section describes how the tradeoffs can be posed as optimization problems for different performance requirements.

Corona-Lite: The first performance goal we set is to minimize the average update detection time while bounding the total network load placed on content servers. Corona-Lite improves the update performance seen by the clients while ensuring that the content servers handle a light load, no more than what they would handle from the clients if the clients fetched objects directly from the servers.

The optimization problem for Corona-Lite is defined in Table 1. The overall update performance is measured by taking an average of the update detection time for each channel weighed by the number of clients subscribed to that channel. We weigh the average using the number of subscriptions because update performance is an end user experience and each client counts as a separate unit in the average. The target network load for this case is simply the total number of subscriptions seen by the system.

Corona-Lite clients experience the maximum benefits of cooperation. Clients of popular channels gain greater benefits than clients of less popular channels. Yet, Corona-Lite avoids suffering from diminishing returns and uses its surplus polling capacity on less popular channels where the extra bandwidth yields higher marginal benefit. Since improvement in update performance is inversely related to the number of polling nodes, there is little benefit in increasing the number of polling nodes beyond a point. A heuristic based scheme that assigns polling nodes in proportion to number of subscribers would clearly suffer from diminishing returns. Corona, on the other hand, distributes the surplus load to other, less popular channels, achieving a better global average update detection time. Consequently, a

less popular channel also gains substantial performance improvement compared to what cooperation between the that channel's clients alone can achieve.

Corona-Lite:

$$\min. \sum_1^M q_i \frac{b^i}{N} \quad \text{s.t.} \quad \sum_1^M s_i \frac{N}{b^i} \leq \sum_1^M q_i$$

Minimize average update detection time, while bounding the load placed on content servers.

Corona-Fast:

$$\min. \sum_1^M s_i \frac{N}{b^i} \quad \text{s.t.} \quad \sum_1^M q_i \frac{b^i}{N} \leq T \sum_1^M q_i$$

Achieve a targeted average update detection time, while minimizing the load placed on content servers.

Corona-Fair:

$$\min. \sum_1^M q_i \frac{\tau}{u_i} \frac{b^i}{N} \quad \text{s.t.} \quad \sum_1^M s_i \frac{N}{b^i} \leq \sum_1^M q_i$$

Minimize average update detection time w.r.t. expected update frequency, bounding load on content servers.

Corona-Fair-Sqrt:

$$\min. \sum_1^M q_i \frac{\sqrt{\tau}}{\sqrt{u_i}} \frac{b^i}{N} \quad \text{s.t.} \quad \sum_1^M s_i \frac{N}{b^i} \leq \sum_1^M q_i$$

Corona-Fair with sqrt weight on the latency ratio to emphasize infrequently changing channels.

Corona-Fair-Log:

$$\min. \sum_1^M q_i \frac{\log \tau}{\log u_i} \frac{b^i}{N} \quad \text{s.t.} \quad \sum_1^M s_i \frac{N}{b^i} \leq \sum_1^M q_i$$

Corona-Fair with log weight on the latency ratio to emphasize infrequently changing channels.

Notation

τ	polling interval
M	number of channels
N	number of nodes
b	base of structured overlay
T	performance target
l_i	polling level of channel i
q_i	number of clients for channel i
s_i	content size for channel i
u_i	update interval for channel i

Table 1: **Performance-Overhead Tradeoffs:** This figure summarizes the optimization problems for different versions of Corona.

Corona-Fast: While Corona-Lite bounds the network load on the content servers and minimizes update latency, the update performance it provides can vary depending on the current workload. Corona-Fast provides stable update performance, which can be maintained steadily at a desired level through changes in the workload. Corona-Fast solves the converse of the above optimization problem; that is, it minimizes the total network load on the content servers while meeting a target average update detection time. Corona-Fast enables us to tune the update performance of the system according to application

needs. For example, a stock-tracker application may pick a target of 30 seconds to quickly detect changes to stock prices.

Corona-Fast shields legacy web servers from sudden increases in load. Sudden increase in the number of subscribers for a channel does not trigger a corresponding increase in network load on the web server, since Corona-Fast does not increase polling after diminishing returns sets in. In contrast, in legacy RSS, popularity spikes cause a significant increase in network load on content providers. Moreover, the increased load typically continues unabated as subscribers forget to unsubscribe, creating “sticky” traffic. Corona-Fast protects content servers from flash-crowds and sticky traffic.

Corona-Fair: Both Corona-Fast and Corona-Lite do not consider the actual rate of change of content in a channel. Update intervals for Web objects are known to vary considerably from a few minutes to no change over several days [10, 19]. Corona-Fair incorporates the update rate of channels into the performance tradeoff in order to achieve a fairer distribution of update performance between channels. It defines update performance as a ratio of the update detection time and the update interval of the channel and aims to minimize the modified metric for a load target.

While the new metric accounts for the wide difference in update characteristics, it biases the performance unfavorably against channels with large update interval times. A channel that does not change for several days experiences long update detection times, even if there are many subscribers for the channel. We correct this bias by exploring other update performance metrics based on square root and logarithm functions, which grow sub-linearly. A non-linear metric dampens the tendency of the optimization algorithm to punish slow-changing yet popular feeds. Table 1 summarizes the optimization problems for different versions of Corona.

3.2 Decentralized Optimization

Corona determines the optimal polling levels using the Honeycomb optimization toolkit. Honeycomb provides numerical algorithms and decentralized mechanisms for solving optimization problems of the following kind on structured overlays.

$$\min. \sum_1^M f_i(l_i) \quad \text{s.t.} \quad \sum_1^M g_i(l_i) \leq T$$

Here, $f_i(l)$ and $g_i(l)$ can define the performance or the cost for channel i as a function of the polling level l . The only restriction imposed by honeycomb is that the functions $f_i(l)$ and $g_i(l)$ are monotonic in l .

Honeycomb’s optimization algorithm runs in $O(M \log M \log N)$ time and is accurate within the

granularity of one channel. Note that the preceding optimization problem is NP Hard as the polling levels only take integral values. Hence, instead of using computationally intensive techniques to find the exact solution, Honeycomb finds an approximate solution quickly in time comparable to a sorting algorithm.

The solution provided by Honeycomb is accurate and deviates from the optimal in at most one channel. Honeycomb achieves this accuracy by finding two solutions that optimize the problem with slightly altered constraint; one with a constraint $T_d \leq T$ and another with constraint $T_u \geq T$. The corresponding solutions L_u^* and L_d^* are exactly optimal for the optimization problems with constraints T_l and T_r respectively, and differ in at the most one channel. That is, one channel has a different polling level in L_l^* than in L_r^* . Honeycomb then chooses L_d^* as the final solution because it satisfies the constraint T strictly.

Honeycomb computes L_d^* and L_u^* by using a Lagrange multiplier to transform the optimization problem as follows:

$$L^* = \arg \min. \sum_1^M f_i(l_i) - \lambda [\sum_1^M g_i(l_i) - T]$$

The monotonicity of $f_i(l)$ and $g_i(l)$ ensures that there is a single minimum over the space of λ . Honeycomb iterates over λ and obtains two solutions L_d^* and L_u^* that bracket the minimum using standard bracketing methods for function optimization in one dimension.

Two observations enable Honeycomb speed up the optimization algorithm. First, $L^*(\lambda')$ for a single iteration can be computed by finding $\arg \min. f_i(l_i) - \lambda' g_i(l_i)$ independently for each channel. This takes $O(M \log N)$ time as the number of levels is bounded by $\lceil \log N \rceil$. Second, for each channel there are only $\log N$ values of λ that change $\arg \min. f_i(l_i) - \lambda' g_i(l_i)$. Pre-computing these λ values for each object provides a discrete iteration space $M \log N$ values. By keeping a sorted list of the λ values, Honeycomb computes the optimal solution in $O(\log M)$ iterations. Overall, the run-time complexity of the optimization algorithm is $O(M \log M \log N)$ time, including the time spent in pre-computation, sorting, and iterations.

The preceding algorithm requires the tradeoff functions $f_i(l)$ and $g_i(l)$ of all channels in the system in order to compute the global optimum. Solving the optimization problem using limited data available locally can produce highly inaccurate solutions. However, collecting the tradeoff factors for all the channels at each node is

¹Note that the optimal solution L^* for the original problem with constraint T may actually decide to replicate objects differently from L_d^* and L_u^* . Yet, the minimum determined by L^* will be bounded by the minima determined by L_d^* and L_u^* due to the monotonicity.

clearly expensive and impractical. It is possible to gather the trade-off data at a central node, run the optimization algorithm in a single location, and distribute the optimal levels to peers from the central location. We avoid using centralized infrastructure as it violates our distributed approach and introduces a single point of failure in the system.

Instead, Honeycomb internally aggregates coarse grained information about global tradeoff factors. It combines channels with similar tradeoff factors into a *tradeoff cluster*. Each cluster summarizes the tradeoff factors for multiple channels and provides coarse-grained tradeoff information. A ratio of performance and cost factors, f_i/g_i is used as a metric to combine channels. For example, channels with comparable values for $\frac{q_i}{u_i s_i}$ are combined into a cluster in Corona-Fair.

Honeycomb nodes periodically exchange the clusters with contacts in the routing table and aggregate the clusters received from the contacts. Honeycomb keeps the overhead for cluster aggregation low by limiting the number of clusters for each polling level to a constant *Tradeoff_Bins*. Each node receives *Tradeoff_Bins* clusters for every polling level from each contact in the routing table. Combined, these clusters summarize the tradeoff characteristics of all the channels in the system. The cluster aggregation overhead in terms of memory state as well as network bandwidth is limited by the size of the routing table, and scales with the logarithm of the system size.

3.3 System Management

Corona is a completely decentralized system, where nodes act independently, share load, and achieve globally optimal performance through mutual cooperation. Corona spreads load uniformly among the nodes through consistent-hashing [18]. Each channel in Corona has a unique identifier and one or more *owner nodes* managing it. The identifier is a content-hash of the channel's URL and the *primary owner* of the channel is the Corona node with the closest identifier to the channel. Corona sets additional owners for a channel in order to tolerate failures. These owners are the f -closest neighbors of the primary owner along the ring. In the event an owner fails, a new neighbor automatically replaces it.

Owners take responsibility for managing subscriptions, polling, and updates for a channel. Owners receive subscriptions through the underlying overlay, which routes all subscription requests of a channel automatically to the node with the closest identifier to the channel's. The owners keep state about the subscribers of a channel and send notifications to them when fresh updates are detected. In addition, owners also keep track of channel specific factors that affect the performance tradeoffs, namely the number of subscribers, the size of the

content, and the interval at which servers update channel content. The latter is estimated based on time between updates detected by Corona.

Corona manages cooperative polling through a periodic protocol consisting of an *optimization phase*, a *maintenance phase*, and an *aggregation phase*. In the optimization phase, Corona nodes apply the optimization algorithm on fine-grained tradeoff data for locally polled channels and coarse-grained tradeoff clusters obtained from overlay contacts. In the maintenance phase, changes to polling levels are communicated to peer nodes in the routing table through *maintenance messages*. Finally, the aggregation phase enables nodes to receive new aggregates of tradeoff factors. In practice, the three phases occur concurrently at a node with aggregation data piggy-backed on maintenance messages.

Corona nodes operate independently and make decisions to increase or decrease polling levels locally. Initially, only the owner nodes at level $K = \lceil \log N \rceil$ poll for the channels. If an owner decides to lower the polling level to $K - 1$ (based on local optimization), it sends a message to the contacts in its routing table at row $K - 1$ in the next *maintenance phase*. As a result, a small wedge of level $K - 1$ nodes start polling for that channel. Subsequently, each of these nodes may independently decide to further lower the polling level of that channel. Similarly, if the home nodes decides to raise the level from $K - 1$ to K it asks its contact in the $K - 1$ wedge to stop polling.

In general, when a level i node lowers the level by $i - 1$ or raises the level from $i - 1$ back to i , it instructs row $i - 1$ in its routing table contacts to start or stop polling for that channel. This control path closely follows the DAG rooted at the owner node. Nodes at level i (depth $K - i$) in this DAG decide whether their children at level $i - 1$ can poll a channel and convey these decisions periodically every *maintenance interval*. When a node begins to start polling for a channel, it waits for a random interval of time between 0 and the polling interval, so that polls for a channel at different nodes are spread over time.

Corona nodes gather current estimates of tradeoff factors in the aggregation phase. Owners monitor the number of subscribers and send out fresh estimates along with the maintenance message. Subsequent maintenance messages sent out by descendant nodes in the DAG carry these estimates to all the nodes in the wedge. Update interval and size of a feed only change during updates and are therefore sent along with updates. Tradeoff clusters are also sent by contacts in the routing table in response to maintenance messages.

Corona inherits its robustness and failure-resilience properties from the underlying structured overlay. A central property of structured overlays is their self-healing

properties around failures. For example, any node in Pastry with i matching prefix digits and j as the $(i + 1)^{th}$ digit can occupy the row i and column j of the routing table. If the current contact in this position fails, the underlying overlay automatically replaces it with another contact satisfying this property. When new nodes join the system or when nodes fail, Corona ensures the transfer of subscription state to the new owners. A node that is no longer an owner simply erases its subscription state and a node that becomes a new owner receives the state from other owners of the channel. Simultaneous failure of more than f adjacent nodes poses a problem for Corona, as well as many other peer-to-peer systems; we assume that f is chosen to make such an occurrence rare. Note that clients can easily renew subscriptions should a catastrophic failure lose some subscription state.

Overall, Corona manages polling using light-weight mechanisms that impose a small, predictable overhead on the nodes and network. Its decision making does not rely on expensive constructs such as consensus, leader election, or clock synchronization. All networking activity is local and limited to contacts in the routing table.

3.4 Update Dissemination

Updates are central to the operation of Corona; hence, we ensure that they are detected and disseminated efficiently. Corona uses monotonically increasing numbers to identify versions of content. The version numbers are based on content modification times whenever the content carries such a timestamp. For other channels, the primary owner assigns version numbers in increasing order based on the updates received by it.

Corona nodes share updates only as *diffs*, the difference between old and new content, rather than the entire content. A measurement study on micronews feeds conducted at Cornell shows that the amount of change in content during an update is typically tiny. The study reports that the average update consists of 17 lines of XML and 6.8% of the content size [19], which implies that a significant amount of bandwidth can be saved through delta-encoding.

A *difference engine* enables Corona to identify when a channel carries new information that needs to be disseminated to subscribed clients. The difference engine parses the HTML or XML content to discover the core content in the channel, ignoring frequently changing elements such as timestamps, counters, and advertisements. The difference engine generates a diff if it detects an update after isolating the core content. The data in a diff resembles the typical output of the POSIX 'diff' command; it carries the line numbers where the change occurs, the changed content, an indication whether it is an addition, omission or replacement, and a version number of the old content to compare against.

When a diff is generated by a node, it shares the update with all other nodes at the same polling level as the channel. To achieve this, the node simply disseminates the diff along the DAG rooted at it up to a depth equal to the polling level of the channel. The dissemination along the DAG takes place using contacts in the routing table of the underlying overlay. For channels that cannot obtain a reliable modification timestamp from the server, the node detecting the update sends the diff to the primary owner, which assigns a new version number and initiates the dissemination to other nodes polling that channel. Two different nodes may detect a change “simultaneously” and send diffs to the primary owner. The primary owner always checks the current diff with the latest updated version of the content and ignores redundant diffs.

3.5 User Interface

Corona employs instant messaging (IM) as its user interface. Users add Corona as a “buddy” in their favorite instant messaging system; both subscriptions and update notifications are then transported as instant messages between the users and Corona. Users send request messages of the form “subscribe url” and “unsubscribe url” to subscribe and unsubscribe for a channel. A subscribe or unsubscribe message delivered by the IM system to Corona is routed to all the owner nodes of the channel, which update their subscription state. When a new update is detected by Corona, the current primary owner sends an instant message with the diff to all the subscribers through the IM system. If a subscriber is offline at the time an update is generated, the IM system buffers the update and delivers it when the subscriber subsequently joins the network.

Delivering updates through instant messaging systems may incur some additional latency, but this latency is typically modest. Instant messaging systems are already designed to reduce such latencies during two-way communication. Moreover, IM systems that allow peer-to-peer communication between their users, such as Skype, do not suffer from the additional latency of tunneling through a centralized service.

Instant messaging enables Corona to be easily accessible to a large user population, as no computer skills other than an ability to “chat” is required. It is freely accessible for users behind public-access computers, which restrict users from reconfiguring the system, as well as users behind fire-walls since instant messaging connections are moderated by centralized services on well-defined ports. Moreover, instant messages also guarantee the authenticity of the source of update messages to the clients, as instant messaging systems pre-authenticate Corona as the source through password verification.

4 Implementation

We have implemented a prototype of Corona as an application layered on Pastry, a prefix-matching structured overlay system [25]. The implementation uses 160-bit SHA-1 hash function to generate identifiers for both the nodes (based on their IP address) and channels (based on their URLs). Both the base of Pastry and the number of tradeoff clusters per polling level are set to 16.

Prefix matching overlays occasionally create *orphans*, that is, channels with no nodes at the baselevel. Orphans can be created because there are no nodes with enough number of matching prefix digits in the system and the required wedge, corresponding to level $\lceil \log N \rceil - 1$ is empty. A consequence of this for Corona is that it cannot assign additional nodes to poll an orphan channel. Left unhandled, this problem can have an adverse impact on the performance tradeoff as update detection times of orphan channels cannot be improved. However, Corona properly handles orphan channels by adjusting the tradeoffs appropriately. The tradeoff factors of orphan channels are aggregated into a *slack cluster*, which is used to correct the performance target prior to optimization.

Corona interacts with IM systems using GAIM [12], an open source instant messaging client for Unix based platforms that supports multiple IM systems including Yahoo Instant Messenger, AOL Instant Messenger, ICQ, and Jabber. Our current implementation uses Yahoo to interface with clients; it is trivial to extend it to other IM systems supported by GAIM and we intend to do so shortly. Yahoo has a limitation that only one instance of a user can be logged on at a time, preventing Corona nodes to be all logged on at the same time. While we hope that Yahoo and other IM systems will support simultaneous logins from automated users such as Corona in the near future, as they have for certain chat robots, our current implementation uses a centralized server to talk to Yahoo as a stop-gap measure. This server acts as an intermediary for all update diffs sent to clients as well as subscription messages sent by clients. Also, Yahoo rate limits instant messages sent by unprivileged clients. Corona's implementation limits the rate of updates sent to clients and avoids sending updates in bursts.

Corona trusts the nodes in the system to behave correctly and generate authentic updates. However, it is possible that in a collaborative deployment, where nodes under different administrative domains are part of the Corona network, some nodes may be malicious and generate spurious updates. This problem can be easily solved if content providers are willing to publish digitally signed certificates along with the content. An alternative solution that does not require changes to servers is to use threshold-cryptography to generate a certificate for content [35, 16]. The responsibility for generating partial

signatures can be shared among the owners on a node ensuring that rogue nodes below the threshold level cannot corrupt the system. Designing and implementing such a threshold-cryptographic scheme is beyond the scope of this paper.

5 Evaluation

We evaluate the performance of Corona through large-scale simulations and wide-area experiments on Planet-Lab [2], a geographically distributed testbed. In all our evaluations, we compare the performance of Corona with the performance of legacy RSS, a widely-used micronews syndication system. The simulations and experiments are driven by real-life RSS traces collected at Cornell.

We collected characteristics of micronews workload and content by passively logging user activity and actively polling RSS feeds [19]. User activity recorded between March 22 and May 3 of 2005 at the gateway of the Cornell University Computer Science Department provided a workload of 158 clients making about 62,000 requests for 667 different feeds. The channel popularity closely follows a Zipf distribution with exponent 0.5. The survey analyzes the update rate of micronews content by actively polling about 100,000 RSS feeds obtained from *syndic8.com*. We poll these feeds at one hour intervals for a duration of 84 hours, and subsequently select a subset of 1000 feeds and poll them at a finer granularity of 10 minutes for 5 days. Comparing periodic snapshots of the feeds shows that the update interval of micronews content is widely distributed; about 10% of channels change within an hour, while 50% of channels did not change at all during 5 days of polling.

5.1 Simulations

We use tradeoff parameters based on the RSS survey. In order to scale the workload to the larger-scale of our simulations, we extrapolate the distribution of feed popularity from the workload traces and set the popularity to follow a Zipf distribution with exponent 0.5. We use distribution for update rate of channels obtained through active polling, setting the update interval of the channels that do not see any updates to one week.

We perform simulations for a system of 1024 nodes, 20,000 channels, and 1,000,000 subscriptions. We start each simulation with an empty state and issue all subscriptions at once before collecting performance data. We run the simulations for six hours with a polling interval of 30 minutes and maintenance interval of one hour. We study the performance of the three schemes, namely, Corona-Lite, Corona-Fast, and Corona-Fair, proposed in Section 1, and compare the performance with that of legacy RSS clients polling at the same rate of 30 minutes.

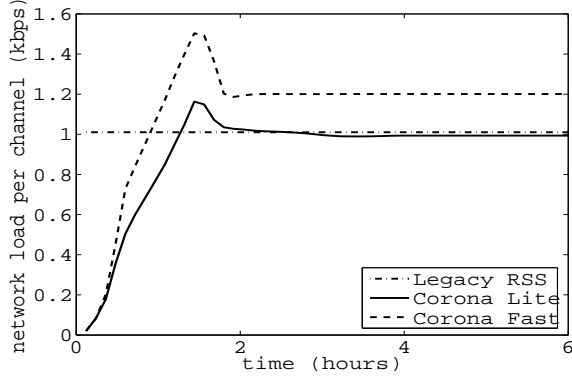


Figure 3: **Network Load on Content Servers: Corona-Lite settles down quickly to match the network load imposed by legacy RSS clients.**

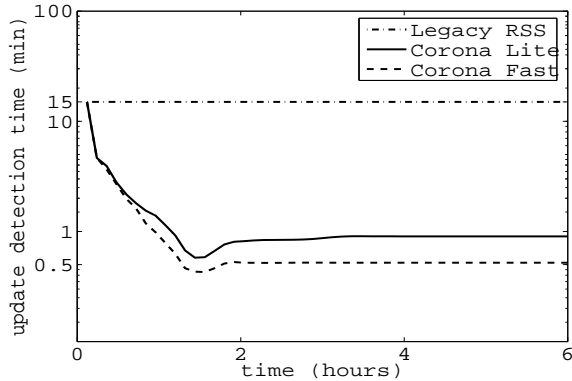


Figure 4: **Average Update Detection Time: Corona-Lite provides 15-fold improvement in update detection time compared to legacy RSS clients for the same network load.**

Corona-Lite

Figures 3 and 4 respectively show the network load and update performance for Corona-Lite, which minimizes average update detection time while bounding the total load on content servers. The figures plot the network load, in terms of the average bandwidth load placed on content servers, and update performance, in terms of the average update detection time. Figure 3 shows that Corona-Lite stabilizes at its target load equal to that imposed by legacy RSS clients. Starting from a clean slate, where only owner nodes poll for each channel, Corona-Lite quickly converges to its target in two maintenance phases. The average load exceeds the target for a brief period before stabilization. This slight delay is due to nodes not having complete information about tradeoff factors of other channels in the system. However, the discrepancy is corrected automatically once global tradeoff factors are aggregated as coarse-grained clusters.

At the same time, Figure 4 shows that Corona-Lite achieves an average update detection time of about one

minute. The update performance of Corona-Lite represents an order of magnitude improvement over the average update detection time of 15 minutes provided by legacy RSS clients. This substantial difference in performance is achieved through judicious distribution of polling load between cooperating nodes, while imposing no more load on the servers than the legacy clients.

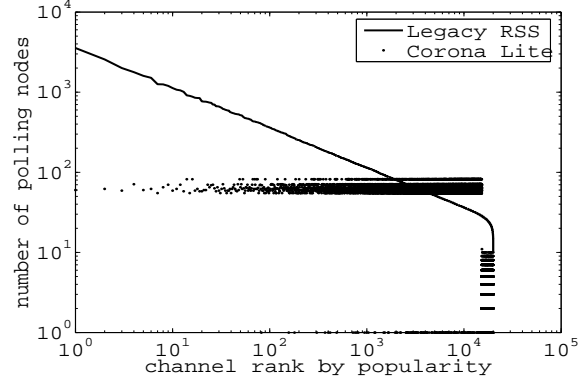


Figure 5: **Number of Pollers per Channel: Corona trades off network load from popular channels to decrease update detection time of less popular channels and achieve a lower system-wide average.**

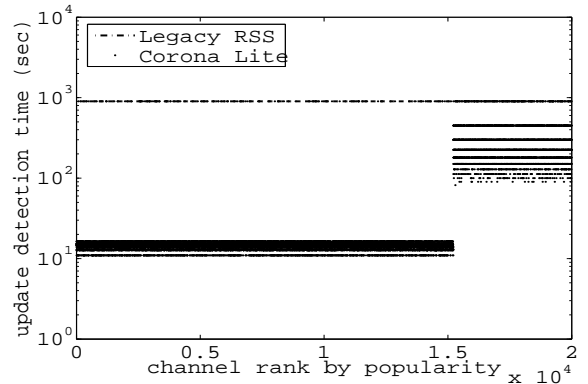


Figure 6: **Update Detection Time per Channel: Popular channels gain greater decrease in update detection time than less popular channels.**

Figures 5 and 6 show the number of polling nodes assigned by Corona-Lite to different channels and the resulting distribution of update detection times. The x-axis shows channels in reverse order of popularity. The load imposed by legacy RSS is equal to the number of clients and, as expected, follows a straight line of slope 0.5 on the log-log plot. Three levels of polling can be identified in Figure 5 for Corona-Lite, channels clustered around 100 at level 1, channels with less than 10 clients at level 2, and orphan channels close to the X-axis with just one owner node polling them. The sharp change in the dis-

tribution after 15,000 channels indicates the point where the optimal solution changes levels.

Figure 5 shows that Corona-Lite favors popular channels over unpopular ones when assigning polling levels. Yet, it significantly reduces the load on servers of popular content compared to legacy clients, which impose a highly skewed load on content servers and overload servers of popular content. Corona-Lite reduces the load at the over-loaded servers, and transfers the extra load to servers of less popular content to improve their update performance.

The above favorable behavior of Corona-Lite is due to diminishing returns caused by the inverse relation between the update detection time and the number of polling nodes. It is more beneficial to distribute the polling across many channels than devote a large percentage of the bandwidth to polling the most popular channel. Nevertheless, load distribution in Corona-Lite respects the popularity distribution of channels; popular channels are polled by more nodes than less popular channels (see Figure 5). The upshot is that popular channels gain an order of magnitude better improvement in update performance than less popular ones (see Figure 6).

Corona-Fast

Figures 3 and 4 show the network load and update performance, respectively, for Corona-Fast, which minimizes the total load on servers while aiming to achieve a target update detection latency. Figure 4 confirms that Corona-Fast closely meets the desired target of 30 seconds. This improvement in update detection time entails an increase in server load over Corona-Lite (see Figure 3). Unlike Corona-Lite, whose update performance may vary depending on the workload seen by the system, Corona-Fast provides a stable average update performance. Moreover, it enables us to set the performance depending on the requirements of the application or users and ensures that the targeted performance is achieved with minimal load on content servers.

Corona-Fair

Finally, we examine the performance of Corona-Fair, which uses update rate of channels to further fine-tune the distribution of load. It takes advantage of the fact that channels with longer update intervals need not be polled as often as rapidly updated channels. Figure 7 shows the distribution of update detection times achieved by Corona-Lite for different channels ranked by their update intervals. Channels with same update intervals are further ranked by popularity. For clarity of presentation, we only plot the distribution for 200 channels, picked at random.

Scheme	Average Update Detection Time (sec)	Average Load (polls per 30 min per channel)
Legacy-RSS	900	50.00
Corona-Lite	54	49.22
Corona-Fair	149	42.65
Corona-Fair-Sqrt	58	49.37
Corona-Fair-Log	55	49.36
Corona-Fast	31	59.44

Table 2: **Performance Summary:** This table provides a summary of average update detection time and network load for different versions of Corona. Overall, Corona provides significant improvement in update detection time compared to Legacy RSS, while consuming the same load.

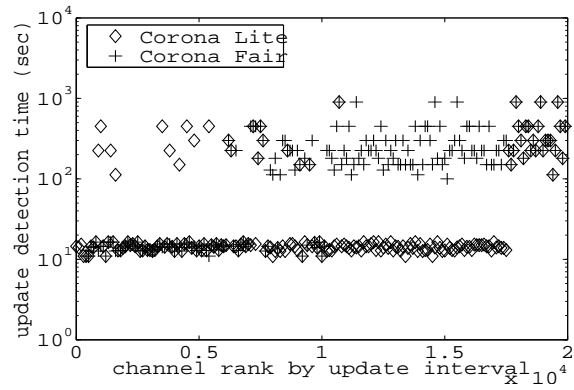


Figure 7: **Update Detection Time per Channel:** Corona-Fair provides greater decrease in update detection time for channels that change rapidly than channels that change rarely.

Figure 7 clearly shows the impact of not using update interval information while assigning polling levels. Channels with large update intervals sometimes have better update detection times (shown in the lower right hand corner) at the expense of rapidly changing channels, which end up with longer update detection times (shown in the upper left hand corner). Corona-Fair fixes the bias by using update intervals of channels to influence polling level optimization. Figure 7 shows that Corona-Fair has a better distribution of update detection times, that is, channels with shorter update intervals have faster update detection time and vice versa.

Corona-Fair, however, introduces a different kind of bias; channels with long update intervals also have long update detection times leading to longer wait times for their clients. This problem can be clearly seen in the right hand side of Figure 7. Section 3.1 proposed two modified metrics based on the square root and logarithm of the update interval to correct this bias. Figure 8 shows the update detection times for these metrics, Corona-Fair-Sqrt and Corona-Fair-Log. Both Corona-Fair-Sqrt and

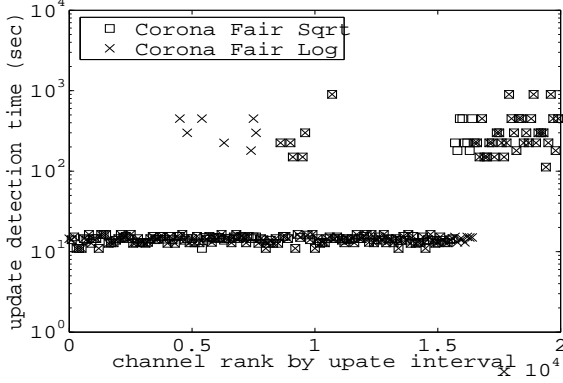


Figure 8: **Update Detection Time per Channel: Corona-Fair-Sqrt and Corona-Fair-Log fix the bias against channels that change rarely and provide better update detection time for them than Corona-Fair.**

Corona-Fair-Log fix the bias introduced by Corona-Fair. In the right hand portion of Figure 8, some channels have faster update detection time, depending on their popularity. Between the two metrics, Corona-Fair-Log is slightly worse than Corona-Fair-Sqrt as in the former case, as some channels with small update interval have long update detection times.

The overall average update detection time and load for different Corona-Fair schemes is shown in Table 2. The average update detection time suffers a little in Corona-Fair compared to Corona-Lite, but the modified Corona-Fair schemes provide an average performance close to that of Corona-Lite.

5.2 Deployment

We deployed Corona on a set of 80 Planet-Lab nodes and measured its performance. The deployment is based on the Corona-Lite scheme, which minimizes update detection time while bounding network load. For this experiment, we have use real channels providing RSS feeds obtained from www.syndic8.com. We issue 30,000 subscriptions for them with a Zipf popularity distribution of exponent 0.5. Subscriptions are issued at a uniform rate during the first one hour of the experiment. The maintenance interval and the polling interval are both set to 30 min. We collected data for a period of six hours.

Figure 9 shows the average update detection time for Corona deployment compared to legacy RSS. Corona decreases the average update time to about 64 seconds compared to legacy RSS. Figure 10 shows the corresponding polling load imposed by Corona on content servers. Corona gradually increases the number nodes polling for objects and reaches a load limit of around 500 loads per minute. This is well shorter than the load imposed by legacy RSS. We have identified this discrepancy to an implementation bug in the mechanism that aggregates to-

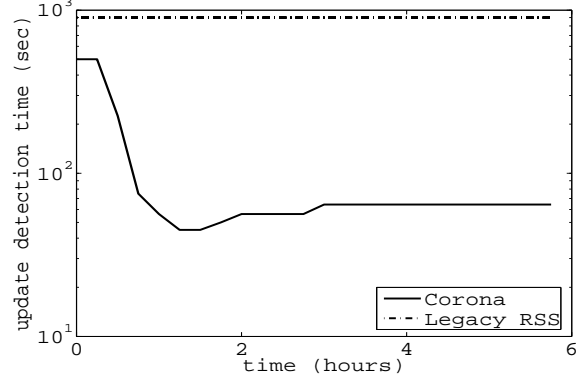


Figure 9: **Average Update Detection Time: Corona provides an order of magnitude lower update detection time compared to legacy RSS.**

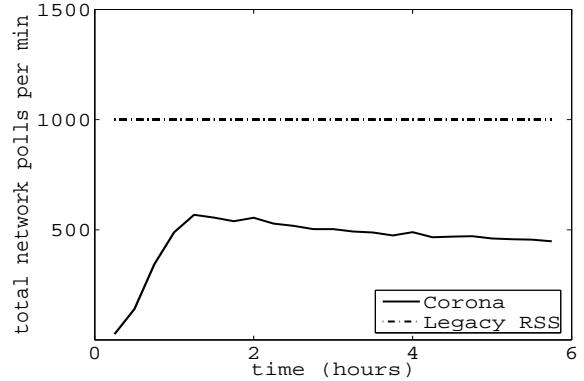


Figure 10: **Total Polling Load on Servers: The total load generated by Corona is well below the load generated by clients using legacy RSS**

tal number of subscriptions made to the system. Nevertheless, we present these graphs to highlight that even while imposing only half the load as legacy RSS, Corona achieves a 15-fold improvement in update detection time.

5.3 Summary

The results from simulations and wide-area experiments confirm that Corona achieves a balance between update latency and network load. It dynamically learns the parameters of the system such as number of nodes, number of subscriptions, and tradeoff factors of all channels, and uses the new parameters to periodically adjust the optimal polling levels of channels and meets performance and load targets. Corona offers considerable flexibility in the kind of performance goals it can achieve. In this section, we showed three specific schemes targeting update detection time, network load, and fair distribution of load under different metrics of fairness. Measurements from the deployment showed that achieving globally optimal performance in a distributed wide-area system is practi-

cal and efficient. Overall, Corona proves to be a high performance, scalable publish-subscribe system.

6 Conclusions

This paper proposes a novel publish-subscribe architecture that is compatible with the existing pull-based architecture of the Web. Motivated by the growing demand for micronews feeds and the absence of any infrastructure to provide asynchronous notifications, we develop a unique solution that addresses the shortcomings of pull based content dissemination and delivers on the promise of a real, deployable, easy-to-use publish-subscribe system.

Corona's unique contribution is the optimal resolution of performance-overhead tradeoffs. Any pull-based content dissemination system has a fundamental tension between the amount of polling required to achieve good update performance and the corresponding network load imposed on content providers. Corona resolves this dilemma by posing the tradeoff as an optimization problem and derives the optimal tradeoff through decentralized, low-overhead mechanisms. Moreover, it provides a "knob" to control the overall performance of the system at fine granularity by setting application-specific performance targets.

Corona's principled approach achieves large gains in performance and scalability. Performance measurements based on simulations and real-life deployment show that Corona clients can achieve several orders of magnitude improvement in update latency. At the same time, Corona bounds the total network load experienced by web servers. Finally, Corona acts as a buffer between clients and servers, shielding servers from the impact of flash-crowds and sticky traffic. Overall, Corona alleviates the twin problems of pull-based systems, namely bad update latencies for clients and high network load on servers, with a single, unified approach.

Corona is currently deployed on Planet-Lab and available for public use. The status of the deployment, including the number of users, channels, and nodes, and the current update performance of Corona, is available online [7]. Overall, we believe that a backwards-compatible, high-performance, efficient publish-subscribe system has the potential to impact how people track frequently changing content on the Web.

Acknowledgements

We would like to thank Rohan Murty for implementing an earlier prototype of the Corona system, and Yee Jiun Song for his help with the system and with his comments on earlier drafts of this paper. This work was supported in part by National Science Foundation Grants 0430161 and CCF-0424422 (TRUST).

References

- [1] Atom. Atom Syndication Format. <http://www.atomenabled.org/developers/syndication>.
- [2] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proc. of Symposium on Networked Systems Design and Implementation*, Boston, MA, Mar. 2004.
- [3] C. Botev and J. Shanmugasundaram. Context Sensitive Keyword Search and Ranking for XML. In *Proc. of International Workshop on Web and Databases*, Baltimore, MD, June 2005.
- [4] L. F. Cabera, M. B. Jones, and M. Theimer. Herald: Achieving a Global Event Notification Service. In *Proc. of Workshop on Hot Topics in Operating Systems*, Elmau, Germany, May 2001.
- [5] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, Apr. 1989.
- [6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.
- [7] Corona. A High Performance Publish-Subscribe System for Web Micronews. <http://www.cs.cornell.edu/people/egs/beehive/corona.php>.
- [8] Y. Diao, S. Rizvi, and M. Franklin. Towards an Internet-Scale XML Dissemination Service. In *Proc. of International Conference on Very Large Databases (VLDB)*, Toronto, Canada, Aug. 2004.
- [9] J. R. Douceur, A. Adya, W. J. Bolosky, and D. Simon. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. In *Proc. of International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.
- [10] F. Douglass, A. Feldman, B. Krishnamurthy, and J. Mogul. Rate of Change and Other Metrics: a Live Study of the World Wide Web. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, Dec. 1997.
- [11] W. Fenner, M. Rabinovich, K. K. Ramakrishnan, D. Srivastava, and Y. Zhang. XTreeNet: Scalable Overlay Networks for XML Content Dissemination and Querying (Synopsis). In *Proc. of International Workshop on Web Content Caching and Distribution*, Sophia Antipolis, France, Sept. 2005.
- [12] GAIM. A Multi-Protocol Instant Messaging Client. <http://gaim.sourceforge.net>.
- [13] B. Glade, K. P. B. R. Cooper, and R. van Renesse. Light-Weight Process Groups in the ISIS System. *Distributed Systems Engineering*, 1(1):29–36, sep 1993.

- [14] Gush. <http://www.2entwine.com>.
- [15] N. Harvey, M. Jons, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proc. of USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, Mar. 2003.
- [16] W. K. Josephson, E. G. Sirer, and F. B. Schneider. Peer-to-Peer Authentication With a Distributed Single Sign-On Service. In *Proc. of International Workshop on Peer-to-Peer Systems*, San Diego, CA, Feb. 2004.
- [17] F. Kaashoek and D. Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. In *Proc. of International Workshop on Peer-to-Peer Systems*, Berkeley, CA, Feb. 2003.
- [18] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigraphy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of ACM Symposium on Theory of Computing*, El Paso, TX, Apr. 1997.
- [19] H. Liu, V. Ramasubramanian, and E. G. Sirer. Client Behavior and Feed Characteristics of RSS, a Publish-Subscribe System for Web Micronews. In *Proc. of ACM Internet Measurement Conference*, Berkeley, CA, Oct. 2005.
- [20] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proc. of ACM Symposium on Principles of Distributed Computing*, Monterey, CA, Aug. 2002.
- [21] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *Proc. of International Workshop on Peer-to-Peer Systems*, Cambridge, CA, Mar. 2002.
- [22] A. Mizrak, Y. Cheng, V. Kumar, and S. Savage. Structured Superpeer: Leveraging Heterogeneity to Provide Constant-time Lookup. In *Proc. of IEEE Workshop on Internet Applications*, San Francisco, CA, Apr. 2003.
- [23] V. Ramasubramanian and E. G. Sirer. Beehive: Exploiting Power Law Query Distributions for $O(1)$ Lookup Performance in Peer-to-Peer Overlays. In *Proc. of Symposium on Networked Systems Design and Implementation*, San Francisco, CA, Mar. 2004.
- [24] S. Ratnasamy, P. Francis, M. Hadley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM*, San Diego, CA, Aug. 2001.
- [25] A. Rowstorn and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg, Germany, Nov. 2001.
- [26] RSS 2.0 Specifications. <http://blogs.law.harvard.edu/tech/rss>, 2005.
- [27] D. Sandler, A. Mislove, A. Post, and P. Druschel. FeedTree: Sharing Web Micronews with Peer-to-Peer Event Notification. In *Proc. of International Workshop on Peer-to-Peer Systems*, Ithaca, NY, Feb. 2005.
- [28] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM*, San Diego, CA, Aug. 2001.
- [29] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An Information Flow Based Approach to Message Brokering. In *Proc. of International Symposium on Software Reliability Engineering*, Paderborn, Germany, Nov. 1998.
- [30] TIBCO Publish-Subscribe. <http://www.tibco.com>.
- [31] TSpaces. <http://www.almaden.ibm.com/cs/TSpaces/>.
- [32] R. van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology For Distributed Systems Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(3), May 2003.
- [33] U. Wieder and M. Naor. A Simple Fault Tolerant Distributed Hash Table. In *Proc. of International Workshop on Peer-to-Peer Systems*, Berkeley, CA, Feb. 2003.
- [34] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.
- [35] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A Secure Distributed On-line Certification Authority. *ACM Transactions on Computer Systems*, 20(4):329–368, Nov. 2002.